DOE RFI Response

Executive summary

Memory safety is a very small part of security. Different application domains have different safety requirements. C++ is a mature language that has proven viable in most application areas and countries. C/C++ is not a language but a style of use. Complete and verified type-and-resource safety is possible in C++. Improved safety — for various safety requirements — is a long-standing tradition in C++. Complete type safety and absence of resource leaks can be had in C++. Changing languages at a large scale is fearfully expensive.

Introduction to C++

We (identified below) are responding to the "Request for Information on Open Source Software Security". https://www.federalregister.gov/documents/2023/08/10/2023-17239/request-for-information-on-open-source-software-security-areas-of-long-term-focus-and-prioritization

C++ has been heavily used in industry and research for decades, and has strived to manage the difficult task of balancing backwards compatibility while evolving continuously with features that improve elegance, reduce errors, and increase expressivity. Backwards compatibility over decades is important when there are many millions of lines of codes dependent on it. Decades-long evolution has made C++23 far safer to use than C++98 (the first standard C++). C++ is also not C. In fact, it comes out of a long-term effort to make C safer and more expressive.

C++ has made great strides in recent years in matters of resource and memory safety [P2687]. C++ benefits from having a formal specification, a fully-specified memory model, and an active community of users and implementers. In contrast, some languages regarded as safe lack a formal specification, which introduces its own safety concerns (e.g., how to ensure a consistent semantic view of code). These

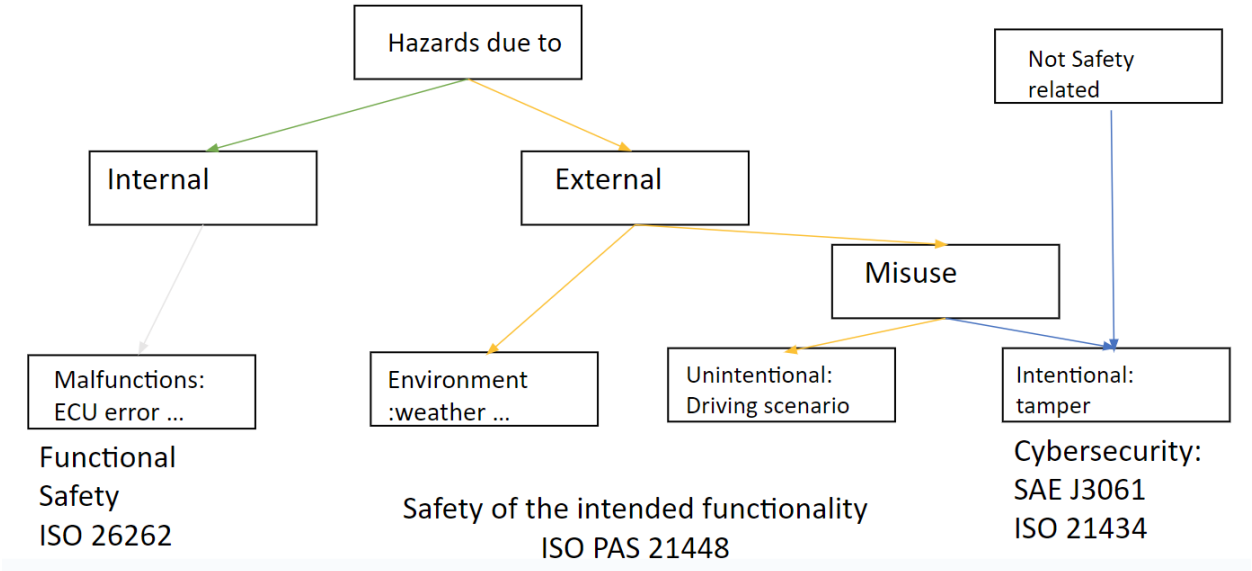
important properties for safety are ignored because the C++ community doesn't have an organization devoted to advertising. C++ is time-tested and battle-tested in millions of lines of code, over nearly half a century, in essentially all application domains. Newer languages are not. Vulnerabilities are found with any programming language, but it takes time to discover them. One reason new languages and their implementations have fewer vulnerabilities is that they have not been through the test of time in as diverse application areas. Even Rust, despite its memory and concurrency safety, has experienced vulnerabilities (see, e.g., [Rust1], [Rust2], and [Rust3]) and no doubt more will be exposed in general use over time.

The important properties of stability, formal specification, applicability in many domains, and ISO process are often ignored because they are not as visibly demonstrable in the language or the compiler (e.g., in the form of safe-by-default semantics or compile-time support for safety).

Technical Aspects

Safety and Security

Safety and security are distinct though they can be dependent in some aspects, but they are often spoken about together as if type safety offered security. ISO defines them differently and handles them in different WGs. Taking the original automotive safety as an example. Functional Safety is handled in ISO 26262 which involves Random faults and Systematic faults. Recently, with the addition of more software in autonomous and semi-autonomous vehicles, ISO has added safety in the form that can be affected by environment as well as unintended misuse under ISO 21448. But deliberate misuse, intrusion and attacks is handled under ISO 21434 or SAE J3061. A separate group handles software over-the-air update faults. This is shown in the following diagram:



We know that large scale applications, or any major systems are often written in multiple languages. It takes at least a decade to rewrite such an application with the collaboration of many teams, passing the torch to the next group as people move in and out. Decades of backwards compatibility also enables a large ecosystem, and reduces the cost of writing new software, allowing an incremental approach. We

know old and new software need to coexist concurrently for some time (at least for the time it takes to build the new system). This is the reality of large scale software design. We are not writing from scratch, which can introduce new vulnerabilities and lead to high cost.

Some concrete examples of safe C++ applications in existence today are in aerospace, automotive, and digital currency.

Safety is not one thing

Language safety is not sufficient, as it compromises other aspects such as performance, functionality, and determinism. What we really want is to have safe features with appropriate guarantees in a language to enable the widest possible domain usage. Another often missed aspect is the library (often called the ecosystem) but in C++ is an integral part of the Standard language delivery. The Library must also be made safe and secure in addition to the language, and there must be a style that allows this safety and security to propagate to user libraries that build on top of the language-supplied library. Indeed, some aspects of safety and security need to be supplied by the Operating System or ideally by hardware.

Different application areas have needs for different kinds of safety and different degrees of safety. For example, the possibility of a resource leak (of memory, file handles, locks, and thread handles) can enable denial of service attacks. Simple lack of data validation is a major source of program misbehavior and security violations. We don't consider a system that relies on manual resource release safe. C++ has offered facilities for handling resource release automatically for decades. Most newer languages do not.

Much of the criticism of C++ is based on code that is written in older styles, or even in C, that do not use the modern facilities aimed to increase type-and-resource safety. Also, the C++ eco system offers a large number of static analysis tools, memory use analysers, test frameworks and other sanity tools.

Fundamentally, safety, correct behavior, and reliability must depend on use rather than simply on language features. We have seen errors costing millions stemming from something as simple as using a less-than rather than a greater-than operator that violates no language rule of any major programming language. Such errors are simply logic errors.

Examples of different kinds of safety that are needed in various industries:

• **Logic errors**: perfectly legal constructs that don't reflect the programmer's intent, such as using operaor < (less than) where a <= or a > was intended.

• **Resource leaks**: failing to delete resources (e.g., memory, file handles, and locks) potentially leading to the program grinding to a halt because of lack of available resources.

• **Concurrency errors**: failing to correctly take current activities into account leading to (typically) obscure problems (such as data races and deadlocks).

• **Memory corruption**: for example, through the result of a range error or by accessing and memory through a pointer to an object that no longer exists thereby changing a different object.

• **Type errors**: for example, using the result of an inappropriate cast or accessing a union through a member different from the one through which it was written.

• **Overflows and unanticipated conversions**: For example, an unanticipated wraparound of an unsigned integer loop variable or a narrowing conversion.

• **Timing errors**: for example, delivering a result in 1.2ms to a device supposedly responding to an external event in 1ms.

• **Allocation unpredictability**: for example, ban on free store allocation "after the engine starts."

• **Termination errors**: a library that terminates in case of "unanticipated conditions" being part of a program that is not allowed to unconditionally terminate.

The C++ Profiles Project (see below) aims to deal with all of those, and more. At its basic level "profiles" borrows its type-and-resource safety profile from the C++ Core Guidelines where it has been tried out:

• Every object is accessed according to the type with which it was defined (type safety)
• Every object is properly constructed and destroyed (resource safety)
• Every pointer either points to a valid object or is the **nullptr** (memory safety)
• Every reference through a pointer is not through the **nullptr** (often a run-time check)
• Every access through a subscripted pointer is in-range (often a run-time check)

Hardware vs. software safety and security

Safety and security in language is also only one component but is one part of the entire programming stack with each part of the stack doing its part

- in the middle of the stack, the programming language itself
- in the upper layers of the stack are modeling languages, frameworks, template libraries, etc.
- in the lower layers are the intermediate languages, drivers, and,
- ultimately, hardware which is increasingly building in safety and cybersecurity notions such as Trusted Execution Environment, and Confidential Computing

We support the idea that the changes for safety and security need to be not just in tooling, but visible in the language/compiler and library. We believe it should be visible such that the "safe and security code" section can be named (possibly using *profiles* — see below), and can mix with traditional code. Individual features may not be very visible, but will be more visible when packaged. This is important for the following reasons

- Intention: This will make it possible for developers and tools to determine intent.
- Visibility: This makes it detectable, most importantly at compile time, but also at run time.
- Composition: This will create composition of profiles, through imports, includes, library calls, and binary inclusions.
    - Not isolated to just software, in some cases, it may be better to do it in hardware
    - Need to corporate with all layers to get the full stack of safety
    - C++ As close to the HW as possible

ISO, maturity, stability, transparency, performance, growth, and backwards compatibility

C/C++, as it is commonly called, is not a language. It is a cheap debating device that falsely implies the premise that to code in one of these languages is the same as coding in the other. This is blatantly false.

C and C++ are two distinct programming languages that share an origin story and have evolved in vastly different directions over the course of decades.

C++ is guided by an international panel of experts representing industry, academia and governments. This consortium operates under the strict guidelines of ISO. Over decades this organization has released several versions of C++: C++98, C++03, C++11, C++14, C++17, C++20 and C++23. Each release builds upon the previous in ways that improve the safety and security of both code that is already written and deployed to the field, and to code that is yet to be written.

It is only through this careful peer-reviewed process that a track record of continual improvement is possible. A few individuals, or a single company, do not have the capacity to consistently achieve all of safety, security and continual improvement of a language over a time span of decades. That is especially the case for a general-purpose language that meets the needs of a diversity of applications ranging from embedded transportation, to finance, to games and entertainment, to extraterrestrial exploration.

The ISO C++ standards committee is the guardian of the stability of the C++ language and standard library. Stability is a major feature. Stability over decades is the only credible guarantee that new code will still work for decades. If some change requires breaking this backwards compatibility, then that needs to be discussed within the whole WG21 community with the input of safety experts. We feel strongly in favor of backwards compatibility of safe code with conventional code.

New languages are always advertised as simpler and cleaner than more mature languages. However, if they succeed, they invariably grow to meet challenges (already addressed in more mature languages). For example, the Java language grew over three times in size, and is better for it.

Education

Many of the problems with C++ code come from poor or outdated education. More education and educational material needs to address safety issues from the start. Teaching unsafe use of pointers and arrays should not be done early on (as is common) and only with suitable warnings. There is no C/C++ language, but there is such usage that enforced language rules would eliminate where needed and certainly in early education.

- Teaching about vulnerabilities and how to avoid them
- Many industry courses, but more need to focus on safety
- Many textbooks though many need to be updated
- The C++ Standard Committee has a study group looking into this (SG20)

Our involvement in safety and security

We are a few C++ senior members with decades experience in ISO C++ (ISO/IEC SC22/WG21) acting as the ISO C++ Directions Group.

We are not strangers to safety and security, with some members having written JSF++[JSF], and participate in some of the key safety ISO, SAE, and UL [UL] groups regarding safety (TC22/SC32 ISO 26262 [SC32], 21448, SAE, UL4600 and AUTOSAR [AUTOSAR]), cybersecurity (ISO 21434), machine

learning safety (SC42 TR5469, PAS8800), MISRA [MISRA], Safety Security Study Group (SG23) in C++, and WG23, as well as C++ Core Guidelines for over a decade. What we have learned is that safety security changes and evolves over time as we learn more and as the industry changes. We believe we should not force safety security on everyone, especially those who don't need or want it. Safety security should not be static, but evolving, as we learn more, and are informed more by outside safety security experts as to what they really need. It is different for different domains which also evolve and change at different rates. For example, aerospace safety is different from medical safety.

While we continue to favor doing more of the early experience with safety concepts in tooling as we have done for decades, we now clearly support safety features in language and library, but packaging several features into *profiles*. We also support pushing/encouraging tools that enable more global analysis to pinpoint safety concerns that are hard for humans to identify.

Profiles Project

To support more than one notion of "safety" and "security", we need to be able to name those notions. We call "profile" a collection of restrictions, requirements, and related semantics that define a safety property to be enforced. A typical profile will not be a simple subset of C++ language features. For example, a range-safety profile cannot simply ban the current unchecked subscripting, but needs to provide a run-time checked alternative for many cases.

Initial work on the idea of profiles can be found in the Core Guidelines (CG, see [C++CG]) and in Section 7 in a recent paper by Stroustrup and Dos Reis [P2687]. Profiles package up several features to make it visible for a code region. Profiles do not limit code in such a way that it reduces the language expressivity like subsets do. We do recognize some domains can deal with subsets and are thus not opposed to a profile-specific subset. However, it is our opinion that subsetting is not a suitable solution for a general purpose language.

Profiles are needed to enforce semantically coherent sets of rules, rather than having individual developers select from a large set of restrictions on individual language features, library facilities, and coding rules.

We like to think profiles do not fragment the ecosystem but increase diversity. Fragmentation occurs when we solve the same problem in different ways. But diversity provides different ways to solve different problems.

We envision that various profiles can appear in source code and automatically trigger analysis. We do not restrict profiles to just address safety concerns, but that they could also support performance concerns, embedded constraints, etc. These cross-cutting aspects can cover different domains: automotive, aerospace, avionics, nuclear, medicine, and so forth. For example we might even have safety profiles for safe-embedded, safe-automotive, safe-medical, performance-games, performance-HPC, and EU-government-regulation.

The set of profiles is open, and only a few would be standardized by WG21 with the rest contributed by industry.

Once we have profiles, we will need to define rules for composition or overriding of different profiles.

Profiles impose restrictions on use where they are activated. They do not change the semantics of a valid program (except to turn UB into a specific well-defined behavior or vice versa). In particular, a piece of code means the same in every profile (or no profiles). This property is the crucial difference between dialects and our approach.

An economic thought experiment

Consider the likely cost of converting a 10M line application from C++ to a different language:

- Needing high reliability and high performance
    - i.e., the kind of system that's critical in some way
- A good developer completes N lines of tested production-quality code a day
    - What is N? 5?, 10?, 100?
    - Say – optimistically – 2,000 lines/year
- Say – optimistically – that a reimplementation (without feature creep) could be 5 million lines
    - Then it would take 500 developers 5 years to complete the new system
    - The old system would have to be maintained for those 5 years say by 50 developers
- What is the loaded salary of a good developer?
    - Employee's compensation plus employers' other cost (e.g., buildings, management structures, computers)
    - Say, $500,000 in the US
    - So, the cost would be 550*5*$500,000 == ~$1,400,000,000
    - Vs. ~$125,000,000 for normal maintenance and development
    - Roughly $1B added cost

- Assumptions
    - US developers
    - developers with relevant experience in the domain and the new language can be found
    - maybe outsourcing could cut cost but would have its own inherent risks
- The new system would be half the size of the old one
    - Better understanding from the start
    - Better language
    - Better tooling
- The new language/languages can cope with the messy parts of the system
- The new system would actually work and be delivered on time
    - Large projects often have time and cost overruns

- There are people for whom $1B or $100M are not scary numbers
- There are people who consider 10M line systems medium sized
    - For a 1M line project, divide by 10; for a 100M-line project (or a set of projects), multiply by 10. There are billions of lines of C++ in current use.
- We consider these numbers an argument for an incremental and evolutionary approach.
    - Obviously, that could be in C++ or in a combination of C++ and other languages
    - Either way, C++ will play a major role

    - We can and must improve C++ — as ever

- A complex system that works is invariably found to have evolved from a simple system that worked.
  — Gall's law

Conclusion

There are economic costs in choosing a language that gives priority to safety and security or is safe-by-default because it pushes the complexity into applications. Not all domains need safety and security, especially high performance computing algorithms. Safety and security should be opt-in instead of by-default.

The other economic concern that has been highlighted above is that we need to consider not just the language, but also the richness of the ecosystem/library, the tooling, the education, and the maturity of all those elements. There should also be an official transparent way to treat issues when they arise, to reduce ambiguity, and to improve safety preferably in an open international standard. C++ has all these aspects enshrined in decades of open public standardization work where hundreds debate in F2F meetings 3 times per year, and thousands discuss its safety and security aspects online. This may give the impression that it is the least safe and secure, when in fact it holds one of the highest positions. The openness of its standard, the transparency of its implementations in open source software are some of what gives it an armor.

There is never a 100% safe language or software, but as long as there is an openness and transparency to allow impartial parties to examine, report, and adjudicate improvements, then there is a chance to make one language safer than another language.

For applications where safety or security issues are paramount, contemporary C++ continues to be an excellent choice.

References

1. [AUTOSAR] https://www.autosar.org/fileadmin/standards/adaptive/18-03/AUTOSAR_RS_CPP14Guidelines.pdf
2. [C++CG] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#pro-profiles
3. [JSF] https://www.stroustrup.com/JSF-AV-rules.pdf
4. [MISRA] https://www.misra.org.uk/misra-c-plus-plus/
5. [P2687] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2687r0.pdf
6. [Rust1] https://blog.g5cybersecurity.com/cve-2020-25792-an-issue-was-discovered-in-the-sized-chunks-crate-through-0-6-2-for-rust/

7. [Rust2] https://blog.sonatype.com/this-week-in-malware-may-13th-edition
8. [Rust3] https://portswigger.net/daily-swig/rust-patches-sneaky-redos-bug
9. [SC32] https://www.iso.org/committee/5383636.html
10. [UL] https://users.ece.cmu.edu/~koopman/ul4600/index.html